mooflow Documentation

Release 0.7b1

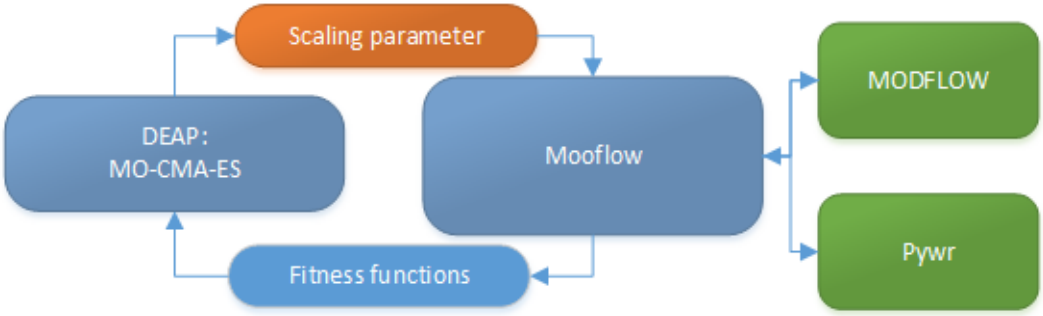
Ruben Müller

Dec 09, 2020

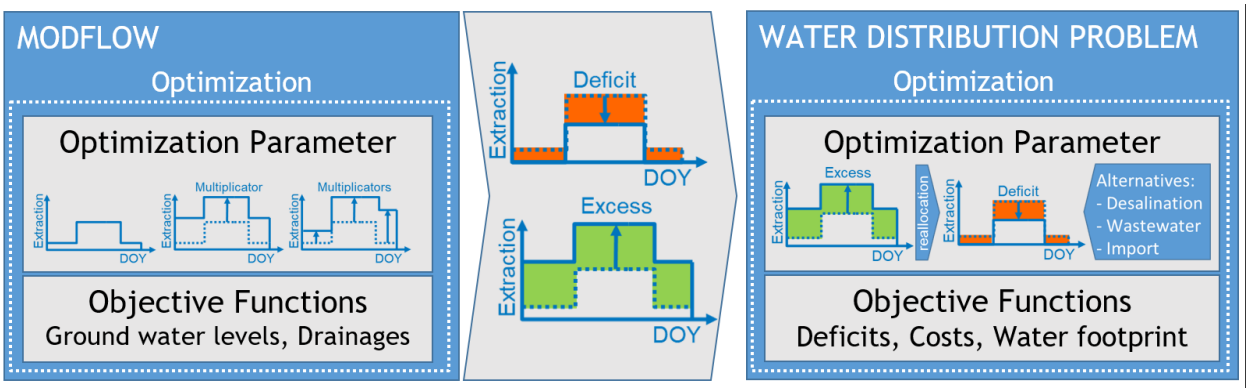
CONTENTS

A generalized framework for basin-scale multi-objective simulation-optimization with MODFLOW

Mooflow helps you to setup and perform multi-objective simulation-based optimization of ground water resources management. Mooflow couples the groundwater model MODFLOW with the water distribution network model Pywr and optimization algorithms from the DEAP package.



Basin-wide optimization can comprise many wells and a substantial number of decision variables. To reduce optimization complexity, Mooflow features methods to minimize the number of decision variable by grouping wells and sharing decision variables. A short technical paper about Mooflow can be found at the Medwater homepage.



CHAPTER

ONE

CODE

Mooflow is available at bitbucket.org.

CHAPTER

TWO

CONTENTS:

2.1 mooflow reference documentation.

2.1.1 Moo_algos

Class to configure and start the MO-CMA-ES optimizer

Mocmaes(config)	Class to setup and run the optimization.

mooflow.moo_algos.Mocmaes

class mooflow.moo_algos.**Mocmaes**(*config*)

Class to setup and run the optimization.

Parameters

config [dict] with the following fields:

"num_parameter" number of optimimzation parameters;

"num_fitness" number of fitness functions;

"pop_size" size of the population (for MOCMAES not so big)

"fitness_weights" list with length num_fitness, -1 for minimization, 1 for maximization;

"target_folder": base folder in which the model folder resides;

"modelname" the name of the model folder

"lower_boundary" lower box constraint (better not change), default 0;

"upper_boundary" upper box constraint (better not change), default 1;

"num_threads" number of threads to use. should exceed the number of physical cores, default: 2;

"init_population" set the parameters for the inidividuals of the initial population, not tested. numpy array with dimension pop x parameter, default False;

"seed" set the seed for the random number generator;

__init__(*config*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(config)	Initialize self.
run()	run the multi-objective optimization with MOC-MAES
setup_modflow_parallel([new_basefolder])	Duplicate the modflow model for parallel evaluation.

Class factory for Mocmaes

create_mocmaes(config)	class factory

mooflow.moo_algos.create_mocmaes

mooflow.moo_algos.**create_mocmaes**(*config*)
class factory

2.1.2 Analyse

Classes

Extr_pattern(ext_times[, ext_rates, multiplier])	The class Pattern describes the inter annual pattern of extraction rate demands for a well.
GW_level_pattern(level_times[, level_gw, ...])	The class Pattern describes the inter annual pattern of extraction rate demands for a well.
Well(idx, times, rates[, multiplier])	
Wellgroup(idx)	
Omo_setup(timestep)	
Area_gw(idx, area_list[, area_list_y, ...])	
Register(reg_type)	
Pywr_model(omo_setup[, timeseries])	
Parameters(parametervector)	

mooflow.classes.Extr_pattern

class mooflow.classes.**Extr_pattern**(*ext_times*, *ext_rates=None*, *multiplier=None*)
The class Pattern describes the inter annual pattern of extraction rate demands for a well.

Parameters

ext_times [list] list with the date of year in which a saison for the extraction pattern ends e.g. [45, 123, 223, 300, 365] for five saisons

ext_rates [list] list with the same length as ext_times gives the extraction rates

multiplier [list] list with the same length as ext_times, sets multiplier for the exttraction rates

__init__(*ext_times*, *ext_rates=None*, *multiplier=None*)
Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`(ext_times[, ext_rates, multiplier])	Initialize self.
`get_pattern_dem`()	
`get_pattern_ext`()	
`get_rate_dem`(time_curr[, year])	Return the demand for the current day of year.
`get_rate_ext`(time_curr[, year])	Return the required extraction rate for the current day of year.
`get_sum_year`([year, demand])	Calculate annual sum of extractions/demands for a year
`set_pattern`(times, rates[, demand])	Set the pattern of required extraction rates and actual demands.

Attributes

`multiplier`	property for the multiplier.

mooflow.classes.GW_level_pattern

class mooflow.classes.**GW_level_pattern**(*level_times*, *level_gw=None*, *multiplier=None*)

The class Pattern describes the inter annual pattern of extraction rate demands for a well.

Parameters

level_times [list] list with the date of year in which a saison for the threshold ends e.g. [45, 123, 223, 300, 365] for five saisons

level_gw [list] list with the same length as ext_times gives the threshold

multiplier [list] list with the same length as ext_times, sets multiplier for the exttraction rates

__init__(*level_times*, *level_gw=None*, *multiplier=None*)

Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`(level_times[, level_gw, multiplier])	Initialize self.
`get_level`(time_curr[, year])	Return the desired ground water level (head) for the current day of year.
`get_levels`()	
`set_pattern`(times, rates)	Set the pattern of required extraction rates and actual demands

mooflow.classes.Well

class mooflow.classes.**Well**(*idx*, *times*, *rates*, *multiplier=None*)

__init__(*idx*, *times*, *rates*, *multiplier=None*)
The class Well describes a well.

Parameters

idx [str] the identificator

times [list] list with the date of year in which a saison for the extraction pattern ends e.g. [45, 123, 223, 300, 365] for five saisons

rates [list] list with the same length as ext_times gives the extraction rates

multiplier [list] list with the same length as ext_times, sets multiplier for the exttraction rates

Methods

__init__(idx, times, rates[, multiplier])	The class Well describes a well.
get_pattern_dem()	
get_pattern_ext()	
get_rate_dem(time_curr[, year])	Return the demand for the current day of year.
get_rate_ext(time_curr[, year])	Return the required extraction rate for the current day of year.
get_sum_year([year, demand])	Calculate annual sum of extractions/demands for a year
is_active_on_date(day, year)	returns True if the date is after the well became active
set_pattern(times, rates[, demand])	Set the pattern of required extraction rates and actual demands.

Attributes

active_on	
location	
multiplier	property for the multiplier.
pywr_node	
sector	
type	

mooflow.classes.Wellgroup

class mooflow.classes.**Wellgroup**(*idx*)

__init__(*idx*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(idx)	Initialize self.
add_wells(well)	add a new well to the group IFF the ID is not already in the list
all_groups()	Returns all IDs of well groups
all_id_groups()	Returns all IDs of well groups
get_settings(setting)	get overview from : lbound, ubound, len_pattern, wells
idx_of_well(wellidx)	return the index of the well in the list
is_in_group(well)	check if the well is in the list
set_group_multiplier(multiplier)	set the multiplier for all wells in the group
set_l_u_boundaries(l_bound, u_bound)	set the multiplier for all wells in the group
set_pywr_node(pywr_node)	set the sector for all wells in the group
set_sector(sector)	set the sector for all wells in the group

Attributes

num_par

mooflow.classes.Omo_setup

class mooflow.classes.**Omo_setup**(*timestep*)

__init__(*timestep*)
Class for the mooflow setup.

Parameters

timestep [int] timestep is the temporal solution with which the model works

Methods

__init__(timestep)	Class for the mooflow setup.
day_of_period(tstep)	Look-up the period for the tstep and determine which day of the period it is.
get_ts_iterator_day()	return an iterator too loop over each timestep in the simulation period
read_disfile([disfile])	Read the discretisation file *.dis.

continues on next page

Table 12 – continued from previous page

run_model([nsilent, report, normal_msg, ...])	This function will run the model using subprocess.Popen.

Attributes

executable
inputfolder
modePywr
modelfolder
modelname
namefile
number_timesteps
setting_time
starting_date
threadnumber
timestep
timestep_lengths
timesteps_end_doy
timesteps_enddates

mooflow.classes.Area_gw

class mooflow.classes.**Area_gw**(*idx*, *area_list*, *area_list_y=None*, *correct_zero=None*)

__init__(*idx*, *area_list*, *area_list_y=None*, *correct_zero=None*)

Class for a ground water level sensitive area.

Parameters

idx [str] the identification of the area

area_list [list] a list of int respresenting the column ids or a list of lists with [column and row ids]

area_list_y [list] needed if area_list is list with column ids

correct_zero [int] add x correct_zero to the column and row ids

Methods

__init__(idx, area_list[, area_list_y, ...])	Class for a ground water level sensitive area.
apply_area_function1(x)	Apply the function that is assigned to the __ares_function.
area_np_array()	returns the x and y coordinates of the area as a numpy array
get_level(time_curr[, year])	Return the desired ground water level (head) for the current day of year.
get_levels()	

continues on next page

Table 14 – continued from previous page

set_area_function1(fargs, *args, **kwargs)	Set a function that evaluated the numpy array that is read from the head file.
set_pattern(times, rates)	Set the pattern of required extraction rates and actual demands

Attributes

area
fitness_group
sector
type
weights
z

mooflow.classes.Register

class mooflow.classes.**Register**(*reg_type*)

__init__(*reg_type*)

Registry for wells, well_groups or ground water level sensitive areas.

Parameters

reg_type: str

well, well_group or area_gw

Methods

__init__(reg_type)	Registry for wells, well_groups or ground water level sensitive areas.
add(obj)	add an object to the register -> only if its ID is not already in the registry
discard(obj)	dicard on object from the Registry
entries_in_rectangle(ul_lr)	
find_well_loc(loc)	find the index of a well, defined by it's coordinates, in the Registry

mooflow.classes.Pywr_model

class mooflow.classes.**Pywr_model**(*omo_setup*, *timeseries=False*)

__init__(*omo_setup*, *timeseries=False*)

Class to manage the water allocation with the Pywr package

Parameters

omo_setup [mooflow.classes.Omo_setup] the configured setup

timeseries [Bool] if False, make a simple balance model (one timestep) if True, take the number of timesteps from the omo_setup

Methods

`__init__`(omo_setup[, timeseries])	Class to manage the water allocation with the Pywr package
`get_active_nodes`()	return the active nodes
`get_json`()	return the dict of the pywr model
`get_pywrfile`()	return the file name of the JSON model file
`nwt_well_shortages`(setup_mf, reg_well)	read the shortages for each well from the list file for the NWT model for each time step for pywr active nodes.
`prepare_pywr_model`(setup_mf, reg_wells, ...)	Return a dictionary which summarizes for the active nodes.
`pywt_extr_nwt`(setup_mf, reg_well, reg_wellgroup)	For active nodes: this function first accumulates the total needed extractions and then accumulates the total shortages in extractions which is read from the list file from the NWT model.
`read_results`([func, return_dict])	Read the results from a pywr model run.
`run_pywr`([use_TablesRecorder])	Start the water allocation optimization pywr.
`set_pywrfile`(file_name[, datafile])	Set the JSON model file for Pywr
`update_nodes_in_model`([factor])	Configure the INFLOW nodes with the deliverable amount of water, e.g.
`write_nodefile`(setup_mf[, delimiter, ...])	Write the CSV data file with all the well data for pywr.

Attributes

`active_nodes`
`limit_zero`
`results`
`usePywr`

mooflow.classes.Parameters

class mooflow.classes.**Parameters**(*parametervector*)

__init__(*parametervector*)
Manage the optimization parameters.

Methods

__init__(parametervector)	Manage the optimization parameters.
rescale_for_wellgroup(group)	Rescale the multiplier for all wells in a wellgroup, using the lower and upper boundary levels in the well group.
rescale_group(group)	Set the multiplier for all wells in the group.

2.1.3 Analyse

Analyse output files

list_file(setup_mf, reg_well)	Read the balance sections of the list file.

mooflow.analyse.list_file

mooflow.analyse.**list_file**(*setup_mf*, *reg_well*)

Read the balance sections of the list file.

Parameters

setup_mf [mooflow.classes.Omo_setup] the modflow setup

reg_well [mooflow.classes.Registry for wells] the registry for wells

Returns

dict [details the balance components]

2.1.4 Opt

Helper functions for the optimizer

distance(feasible_ind, original_ind)	A distance function to the feasibility region.
closest_feasible(individual)	A function returning a valid individual from an invalid one.
valid(individual)	Determines if the individual is valid or not.

mooflow.opt.distance

mooflow.opt.**distance**(*feasible_ind*, *original_ind*)

A distance function to the feasibility region.

mooflow.opt.closest_feasible

mooflow.opt.**closest_feasible**(*individual*)
A function returning a valid individual from an invalid one.

mooflow.opt.valid

mooflow.opt.**valid**(*individual*)
Determines if the individual is valid or not.

2.1.5 Params

Helper functions to handle optimization parameters

check_boundary_len(boundary, pattern_a)	check the length of a boundary list against an pattern.
check_lu_range(bound_l, bound_u)	raise an ValueError if a value in the lower boundary list is higher then in the upper boundary list
recalc_parameter(parameter, bound_l, bound_u)	Rescale the parameter list from [0,..., 1] into the range given by the lower and upper boundary lists.

mooflow.params.check_boundary_len

mooflow.params.**check_boundary_len**(*boundary*, *pattern_a*)
check the length of a boundary list against an pattern.

Parameters

boundary: float or Iterable the upper or lower boundary

pattern_a [Extr_pattern object] The extraction pattern

Returns

boundary [list] the upper or lower boundary. If an Int was provided, return a list with the appropiate length

mooflow.params.check_lu_range

mooflow.params.**check_lu_range**(*bound_l*, *bound_u*)
raise an ValueError if a value in the lower boundary list is higher then in the upper boundary list

mooflow.params.recalc_parameter

mooflow.params.**recalc_parameter**(*parameter*, *bound_l*, *bound_u*)
Rescale the parameter list from [0,..., 1] into the range given by the lower and upper boundary lists.

Parameters

parameter [int or list] the scaled parameter [0,..., 1]

bound_l [Iterable] the lower boundary

bound_u [Iterable] the upper boundary

Returns

parameter [list] the rescaled parameter (multiplicator)

2.1.6 Read

Read MODLFOW model output

read_well_file(registry, timestep[, ts_format])	
wells_and_wellgroups(setup_mf, ...[, ...])	creates registries with the wells and well_goups.
boundaries(boundfile, reg_wellgroup[, ...])	read the lower and upper boundaries from a csv file
read_b_heads_reg(registry, setup_mf[, ...])	Read the heads for specified regions, timestep times and apply a function to integrate the heads
lstfile_iter_well_drn(lst_file[, marker, ...])	Generator function that scans the output of readlines on the results file *.LIST to return an numpy array dtype float' with the results for the current time step
lstfile_iter_balance(setup_mf[, ...])	Generator function that scans the output of readlines on the results file *.LIST to return an numpy array dtype float' with the results for the current time step
lstfile_nwt_shortage(lst_file[, lines_after_m1])	Generator function that scans the output of readlines on the results file *.LIST to return an numpy array dtype float with the results for the current time step

mooflow.read.read_well_file

mooflow.read.**read_well_file**(*registry*, *timestep*, *ts_format=4*)

mooflow.read.wells_and_wellgroups

mooflow.read.**wells_and_wellgroups**(*setup_mf*, *wellgroupfile*, *patternfile*, *delimiter_wg=';'*, *delimiter_pf=';'*, *factor=1*, *timesteps=None*)

creates registries with the wells and well_goups.

Returns

list with the register for wellgroup and wells

mooflow.read.boundaries

mooflow.read.**boundaries**(*boundfile*, *reg_wellgroup*, *delimiter=';'*, *factor=1*)

read the lower and upper boundaries from a csv file

Parameters

boundfile [str] the csv file with the boundary information

reg_wellgroup [mooflow.classes.Registry object] the registry for the wellgroups

delimiter [str, optional] the delimiter used in the boundfile

mooflow.read.read_b_heads_reg

mooflow.read.**read_b_heads_reg**(*registry*, *setup_mf*, *timesteps=None*, *critical=False*)
Read the heads for specified regions, timestep times and apply a function to integrate the heads

Parameters

registry [mooflow.classes.Registry object] the registry with the locations

setup_mf [mooflow.classes.Omo_setup object] the model setup

timesteps [int or list of int, optional] the timesteps to read. CAUTION: timesteps counting start with 0. if timesteps is None, then the heads for all timesteps of the simulation run (as provided by the setup_mf) are provided

Returns

head_sect [dict] a dictionary with entries of region.ids and subentries of Z, if needed

mooflow.read.lstfile_iter_well_drn

mooflow.read.**lstfile_iter_well_drn**(*lst_file*, *marker='WEL_'*, *nrows=477*, *ncols=4*, *tstep=0*)
Generator function that scans the output of readlines on the results file *.LIST to return an numpy array dtype float' with the results for the current time step

Parameters

lst_file [str or list of str] the results file as read with readlines() or the path to the file

marker [str] the first part of the fil,

tstep [int] number of first time step –> the identifier of the WELL/DRN file is build from this

mooflow.read.lstfile_iter_balance

mooflow.read.**lstfile_iter_balance**(*setup_mf*, *period_number=1*, *style='numpyarray'*)
Generator function that scans the output of readlines on the results file *.LIST to return an numpy array dtype float' with the results for the current time step

Parameters

lst_file [Omo_setup] the results file as read with readlines() or the path to the file

tstep [int] number of first time step –> the identifier of the WELL/DRN file is build from this

style [str] numpyarray or dict

Returns

an iterator with returns a numpy array if style is numpyarray or a dictionary if syle is dict

mooflow.read.lstfile_nwt_shortage

mooflow.read.**lstfile_nwt_shortage**(*lst_file*, *lines_after_m1=None*)

Generator function that scans the output of readlines on the results file *.LIST to return an numpy array dtype float with the results for the current time step

Parameters

lst_file [str or list of str] the results file as read with readlines or the path to the file

lines_after_m1 [int] lines offset "OUTPUT CONTROL FOR STRESS PERIOD

2.1.7 Funcs

Functions to comupte fitness functions or indicators

linear_slope(x[, plot_flag, begin])	Get the slope from a linear regression.
days_violated(x, restrictions, setup_mf[, ...])	calculates the longest consecuate period with a violation of a restriction
max_len_ones(inarray[, indicator, ...])	run length encoding.
sum_heads_fg(dict_heads, fitness_group)	calculates the weighted sum of all the indicators in the head dict for one fitness group.
count_neg_slopes(dict_heads, fitness_group)	count the number of area_gw with negative slopes –> depletion trend
sum_neg_slopes(dict_heads, fitness_group[, ...])	sum the negative slopes of selected area_gw, if we have an positive slope, then assign zero –> minimize the depletion trends, but do not recommend positive trends.
pywr_total__no_recorder_noparameters(pywr_model)	compute the deficits, provided rates for total balance runs (no time series, no input data time series written.)

mooflow.funcs.linear_slope

mooflow.funcs.**linear_slope**(*x*, *plot_flag=False*, *begin=0*)

Get the slope from a linear regression.

mooflow.funcs.days_violated

mooflow.funcs.**days_violated**(*x*, *restrictions*, *setup_mf*, *start_with=None*)

calculates the longest consecuate period with a violation of a restriction

Parameters

x [numpy array] time series with the groundwater levels

restrictions [GW_level_pattern] holds the grounwater level restrictions

setup_mf [Omo_setup] the optimization setup

start_with [list] no is done evaluation before [year, doy]

Returns

int [number of timesteps in the longest running violation period]

mooflow.funcs.max_len_ones

mooflow.funcs.**max_len_ones**(*inarray*, *indicator='max_len'*, *parameter=14*, *fitness_group=None*)
run length encoding. If this function is set for a *set_area_function1*, then omit the inarray.

Adopted from: https://stackoverflow.com/questions/1066758/find-length-of-sequences-of-identical-values-in-a-numpy-array-run

Parameters

inarray [numpy array or list] a 1d list or array with ones for violations and zeros for comliances with the gwr pattern

indicator [str] the type of evaluation: *max_len...* the maximum consecuative days violating the constraint *num_days_th...* the longest period with consecuative violations over a treshold *perceptile...* returns a quantile of days with violation

parameter [int] the function of the parameter depends on the indicator *num_days_th...* number of allowed consecuative days with violations *perceptile...* the percentile

mooflow.funcs.sum_heads_fg

mooflow.funcs.**sum_heads_fg**(*dict_heads*, *fitness_group*)
calculates the weighted sum of all the indicators in the head dict for one fitness group.

Parameters

dict_heads [dict] the heads dict / heads evaluated with areal functions for locations

fitness_group [str] the desired fitness group

Returns

int [the weighted sum of the indicators]

mooflow.funcs.count_neg_slopes

mooflow.funcs.**count_neg_slopes**(*dict_heads*, *fitness_group*)
count the number of area_gw with negative slopes –> depletion trend

Parameters

dict_heads [dict] the heads dict / heads evaluated with areal functions for locations

fitness_group [str] the desired fitness group

Returns

int [the weighted sum of the indicators]

mooflow.funcs.sum_neg_slopes

mooflow.funcs.**sum_neg_slopes**(*dict_heads*, *fitness_group*, *multiplier=1*)
sum the negative slopes of selected area_gw, if we have an positive slope, then assign zero –> minimize the depletion trends, but do not recommend positive trends.

Parameters

dict_heads [dict] the heads dict / heads evaluated with areal functions for locations

fitness_group [str] the desired fitness group

multiplier [float] multiplies with the negative slopes.

Returns

——-

int [the weighted sum of the indicators]

mooflow.funcs.pywr_total__no_recorder_noparameters

mooflow.funcs.**pywr_total__no_recorder_noparameters**(*pywr_model*)
comput the deficits, provided rates for total balance runs (no time series, no input data time series written.)

2.1.8 Util

Functions for dates

datetime_ymd(dt)	convert [doy, year] into datetime.datetime
dy_datetime(dt)	
is_leap_year(year)	if year is a leap year return True else return False
doy(Y[, M, D])	given year, month, day return day of year Astronomical Algorithms, Jean Meeus, 2d ed, 1998, chap 7
woy(time_curr[, year])	given the current datetime or doy and year, lookup the week of the year
ymd(Y[, N])	given year = Y and day of year = N, return year, month, day Astronomical Algorithms, Jean Meeus, 2d ed, 1998, chap 7
full_steps_timediff(d1, d2[, settingtime])	calculate the timesteps in weeks, months and days between 2 dates.
timestep_gen(cls)	yields a timestep
days_dt(dt)	calculate the number of days for any month
days_in_month(dt)	calculate the number of days for any month for individual (year, dayOfYear), datetime.datetime or lists of these
stressperiod_day(**kwargs)	returns a list with the number of days for each stress period.

mooflow.utils.datetime_ymd

mooflow.utils.**datetime_ymd**(*dt*)
convert [doy, year] into datetime.datetime

mooflow.utils.dy_datetime

mooflow.utils.**dy_datetime**(*dt*)

mooflow.utils.is_leap_year

mooflow.utils.**is_leap_year**(*year*)
if year is a leap year return True else return False

mooflow.utils.doy

mooflow.utils.**doy**(*Y*, *M=1*, *D=1*)
given year, month, day return day of year Astronomical Algorithms, Jean Meeus, 2d ed, 1998, chap 7

mooflow.utils.woy

mooflow.utils.**woy**(*time_curr*, *year=None*)
given the current datetime or doy and year, lookup the week of the year

mooflow.utils.ymd

mooflow.utils.**ymd**(*Y*, *N=None*)
given year = Y and day of year = N, return year, month, day Astronomical Algorithms, Jean Meeus, 2d ed, 1998, chap 7

mooflow.utils.full_steps_timediff

mooflow.utils.**full_steps_timediff**(*d1*, *d2*, *settingtime=0*)
calculate the timesteps in weeks, months and days between 2 dates.

Parameters

d1 [datetime or iterable] the starting date as datetime or as [doy, year]

d2 [datetime or iterable] the ending date as datetime or as [doy, year]

Returns

dict [dict] the number of timesteps for month (key 12), week (key 7) and days (key 1)

mooflow.utils.timestep_gen

mooflow.utils.**timestep_gen**(*cls*)
yields a timestep

Parameters

starting_date [list] consists of two int [day of year, year]

timesteps [int] the number of time steps in the simulation

timestep [int] the temporary resolution. 1: day, 7: week, 12: month

Returns

tmp [list] consists of [timestep counter, [current day of year, current year]]

mooflow.utils.days_dt

mooflow.utils.**days_dt**(*dt*)
calculate the number of days for any month

Parameters

dt [list, datetime.datetime] list with (year, dayOfYear) or datetime.datetime object to look up

Returns

int [the number of days]

mooflow.utils.days_in_month

mooflow.utils.**days_in_month**(*dt*)
calculate the number of days for any month for individual (year, dayOfYear), datetime.datetime or lists of these

Parameters

dt [list, datetime.datetime] list with (year, dayOfYear) or datetime.datetime object to look up

Returns

int of list of int: the number of days

mooflow.utils.stressperiod_day

mooflow.utils.**stressperiod_day**(***kwargs*)
returns a list with the number of days for each stress period. A stress period is a month. The length of the returned list is the number of total stress periods in the simulation.

Parameters

****kwargs** [. . .] 'datetime' : list with datettime of starttime and simulation length in days

'doy': list with list(year, doy) and simulation length in days

'setup': omo.classes.Omo_setup object

Returns

list with the number of days for each stress period (month) and, if extended, plus the beginning of month

Utility functions

class_by_bisect(value, ranges[, direction])	bisect an list for a value and return the position
make_shape_two(item[, date, length, filler])	returns a string with a given length.
load_obj_compressed(name[, ending])	Loads a prickle binary file.
save_obj(name, obj[, ending])	Save a struct or list into a prickle binary file.
load_obj(name[, ending])	Load a prickle binary file.

mooflow.utils.class_by_bisect

mooflow.utils.**class_by_bisect**(*value*, *ranges*, *direction='left'*)
bisect an list for a value and return the position

Parameters

value [int or float] value to look-up

ranges [list] look-up list needs to be sorted

Returns

position [int] the right side position

mooflow.utils.make_shape_two

mooflow.utils.**make_shape_two**(*item*, *date=None*, *length=2*, *filler='0'*)
returns a string with a given length. padding with 0. – if item is a datetime-object, then date must be given. – if item is a int or str, then date is None

Parameters

item [datetime-object or str / int] the str to be padded

date [string] unit (month, year, day, minute, second)

length [int] length of output string

mooflow.utils.load_obj_compressed

mooflow.utils.**load_obj_compressed**(*name*, *ending='pklz'*)
Loads a prickle binary file.

Parameters

name [string] path and filename for prickle file

mooflow.utils.save_obj

mooflow.utils.**save_obj**(*name*, *obj*, *ending='pkl'*)
Save a struct or list into a prickle binary file.

Parameters

name [string] path and filename for prickle file without file ending, functions add .pkl

obj [struct, list,...] object to save in prickle file

mooflow.utils.load_obj

mooflow.utils.**load_obj**(*name*, *ending='pkl'*)
Load a prickle binary file.

Parameters

name [string] path and filename for prickle file

2.1.9 Write

Write well files

write_well_day(setup_mf, well_reg)	write the pattern with the well files on a daily basis.
write_well_dinm(setup_mf, well_reg)	write the pattern with the well files for each month of the year.

mooflow.write.write_well_day

mooflow.write.**write_well_day**(*setup_mf*, *well_reg*)
write the pattern with the well files on a daily basis.

Parameters

setup_mf [mooflow.classes.Omo_setup] the modflow setup

well_reg [mooflow.classes.Registry for wells] the registry for wells

mooflow.write.write_well_dinm

mooflow.write.**write_well_dinm**(*setup_mf*, *well_reg*)
write the pattern with the well files for each month of the year.

Parameters

setup_mf [mooflow.classes.Omo_setup] the modflow setup

well_reg [mooflow.classes.Registry for wells] the registry for wells

2.2 Tutorial

This example is code (slightly modified) used for the Medwater project (see the technical paper here).

2.2.1 Python Code

Create the main script that configures and duplicates the optimization environment, then runs the optimization algorithm.

```
import mooflow
from eval_modflow import eval_modflow
configuration = {
     "num_parameter": 10,                    # number of optimization parameters
     "eval_function": eval_modflow,  # number of optimization parameters
     "num_threads": 12,                              # parallelisation using 12␣
↪threads
     "fitness_weights": [-1, -1],     # -1 to minimize a fitness function
     "fitnessfile_with_path": "d:/modflow_model/eval_modflow.py"
     }

opt_mocmaes = mooflow.create_mocmaes(configuration)  # initialize the optimization␣
↪algorithm
mocmaes.setup_modflow_parallel(configuration)            # duplicate the model for␣
↪parallelization
mocmaes.run()                                            # run the optimization
```

The eval_modflow.py as mentioned in fitnessfile_with_path looks like this:

```
# -*- coding: utf-8 -*-
from datetime import timedelta
from datetime import datetime
import mooflow as omo
import numpy

def external_modflow(parameters, threadnumber):

    ############################
    # model setup
    ############################

    # working directory for external model
    mf = "d:\\medwater_2050_rri\\Model_2020__" + str(threadnumber)
    # some supplementary files are here
    mf2 = 'd:\\medwater\\MODEL_FILES'
    # directory for output files (log output files with parameter and fitness␣
↪function values go here)
    mf3 = 'd:\\medwater_2050_rri\\'
    # name of modflow executable
    exe = "d:\\medwater_2050_rri\\Model_2020__" + str(threadnumber) + "\\MODFLOW-NWT_
↪64.exe"
    # well file: specifications for each well (x, y, z, name, wellgroup, active node,␣
↪pattern class)
    wellgroupfile = 'd:\\medwater\\MODEL_FILES\\wells_list.txt'
    # extraction pattern file: 16 generalized extraction patterns
    patternfile = 'd:\\medwater\\MODEL_FILES\\pattern_16_inf.txt'
    # boundary file: the upper and lower boundaries for the patterns of each well
    boundfile = 'd:\\medwater\\MODEL_FILES\\boundaries_neu.csv'
    # pywr model file
    pywr_file = 'd:\\medwater\\MODEL_FILES\\nc_2020_des_m50__2050_rri.json'

    # create the model setup
    setup_mf = omo.classes.Omo_setup('month')
```

(continues on next page)

(continued from previous page)

```
    setup_mf.modelfolder = mf
    setup_mf.usePywr = True
    setup_mf.modePywr = "total"
    setup_mf.modelname = 'WMA'
    setup_mf.starting_date = datetime(1987, 9, 15)

    # number of periods to skip for model set in
    setup_mf.setting_time = 2
    # read the temporal discretization from the dis file
    setup_mf.read_disfile()
    # sub-directory for input data
    setup_mf.inputfolder = 'ref'
    setup_mf.executable = exe


    ############################
    # Parameter class
    ############################
    Parameter = omo.classes.Parameters(parameters)

    ############################
    # registry for well groups and wells
    # several wells are grouped and share the same parameter(s)
    # which scale the extraction/infiltration pattern
    # pattern and upper/lower bounds for scaling
    # are read from a binary file
    ############################
    reg_wellgroup, reg_well = omo.read.wells_and_wellgroups(setup_mf,
       wellgroupfile, patternfile, delimiter_wg="\t", delimiter_pf="\t",
       factor=1)
    reg_wellgroup = omo.read.boundaries(boundfile, reg_wellgroup, factor=1)

  ############################
  # setup the Pywr water allocation model
  # the models configuration is stored in a json file
  # active nodes are the sectoral demands
  ############################
    Pywr_model = omo.classes.Pywr_model(setup_mf, timeseries=False)
    Pywr_model.set_pywrfile(pywr_file, mf2)
    Pywr_model.active_nodes = ['a214_IS', 'a210_IS',
            'a210_WB', 'a211_IS',
                'a211_WB', 'a212_IS',
                'a212_WB', 'a220_IS',
                'a220_WB', 'a221_WB',
                'Inf_210IS', 'Inf_211IS']

    ############################
    # ground water level sensitive locations
    # locations are areas which are set by the row/col grid numbers
    # or each area an median is calculated over all grid points-
    # finally an indicator can be selected
    # which can later be used in a fitness function
    ############################
    reg_argw = omo.classes.Register('Area_gw')
    well_file0 = omo.read.read_well_file(setup_mf, 1)
    well_locations = [
        [1, 41, 93],
```

(continues on next page)

(continued from previous page)

```
        [1, 124, 71],
        [1, 310, 51],
        [1, 234, 54],
        [1, 79, 73],
        [1, 92, 106],
        [1, 172, 80],
        [2, 158, 101],
        [3, 197, 115],
        [3, 267, 103]
    ]
   well_thresholds = [9, 12, 13, 13, 13, 20, 18, 35, 295, 345]

    for i, j in enumerate(well_locations):
            for fitness_gwl in range(2, 4):
                    loc = omo.classes.Area_gw('loc' + str(i) + "_" + str(fitness_
↪gwl),
                                      [[j[ 1], j[2]],
                                      [j[1]-1, j[2]],
                                      [j[1]+1, j[2]],
                                      [j[1], j[2]-1],
                                      [j[1], j[2]+1]])

            if fitness_gwl == 0:
                    # calculate 0.5 quantile of area
                    loc.set_area_function1(numpy.percentile, 50, axis=1)
            elif fitness_gwl == 1:
                    # max. len of all periods with ground water level below threshold
                    loc.set_area_function1(omo.funcs.max_len_ones, indicator="max_len
↪", parameter=1)  ### BUG!
            elif fitness_gwl == 2:
                    # total number of days whith a ground water level  below the␣
↪threshold
                    loc.set_area_function1(omo.funcs.days_violated, loc, setup_mf)
            else:
                    # linear trend in ground water level (times -1) if the slope is␣
↪negative, else 0
                    loc.set_area_function1(omo.funcs.linear_slope, plot_flag=0)

            # the treshold in gwl can be a pattern or is constant
            loc.set_pattern([365], [well_thresholds[i]])

            # for several z layers at a location set weights for each layer
            fg = 'redlines' if fitness_gwl == 2 else 'slopes'
            loc.weights = 1
            loc.z = j[0]
            loc.fitness_group = fg
            reg_argw.add(loc)  # register the ground water level sensitive location


    ############################
    # MOCMAES algorithm is "scale sensitive"
    # all parameters are handled between 0 and 1.
    # we transform the parameters back from scaled range [0, ..., 1]
    # --> we rescale the extractions for real word usage in MODFLOW
    ############################
    for _, wg in enumerate(reg_wellgroup):
        Parameter.rescale_for_wellgroup(wg)
```

(continues on next page)

(continued from previous page)

```
    ############################
    # write the model configuration for wells
    ############################
    omo.write.write_well_dinm(setup_mf, reg_well)


    ############################
    # we can now evaluate the parameter vector with MODFLOW.
    # call the local copy of MODFLOW in the subfolder ending with the right tread
    ############################
    setup_mf.run_model(silent=True)

    ############################
    # read results
    ############################

    # read the heads file and return a dictionary with the
    heads for all area_gw
    dict_heads = omo.read.read_b_heads_reg(reg_argw, setup_mf)

    ############################
    # analyse lst file
    ############################

    # read the balance section from the list file / get time series for drains and␣
↪storage
    dict_lst = omo.analyse.list_file(setup_mf, reg_well)

    ############################
    # prepare Pywt model
    ############################

    # calculate the available water for input nodes, read shortages in pumping␣
↪(MODFLOW NWT model required)
    dict_node_provides = Pywr_model.pywt_extr_nwt(setup_mf,
        reg_well, reg_wellgroup)

    ############################
    # prepare and run Pywt model
    ############################

    # prepare the pywr model and run it
    Pywr_model.update_nodes_in_model(1/20) # work with mean values for the 20 year␣
↪perios
    Pywr_model.run_pywr()

    ############################
    # analyse the model
    ############################

    # read the results from the pywr model
    Pywr_model.read_results()

    ############################
    # compile the fitness functions
    ############################
```

(continues on next page)

(continued from previous page)

```
    # the number of consecutive days with a drainage outflows (springs) under␣
↪threshold
    ind_drain = omo.funcs.max_len_ones(dict_lst['drains array'] > 123456,
            indicator="max_len")

    # integrate the indicators for the heads --> no negative slope.
    # ~ for sustainability we want the trends in the groundwater levels over time
    # to be as close to zero or positive
    ff_eco = ind_drain \
        + omo.funcs.sum_neg_slopes(dict_heads, 'slopes', multiplier=1000) \
        + sum([dict_heads[x]["values"] for x in dict_heads if dict_heads[x]["fitness_
↪group"] == "redline"])

    # the weighted sum of all deficits of sectoral demands
    # (1) deficits occur when the pumping rate was set too low (scaling of the␣
↪extraction pattern)
    # or if (2) the realized extraction in MODFLOW was not sufficient or
    # (3) available alternative water sources are insufficient
    ff_shrt = sum([Pywr_model.results['Agr_210IS']['deficit_percent'],
            Pywr_model.results['Agr_211IS']['deficit_percent'],
                Pywr_model.results['Agr_212IS']['deficit_percent'],
                Pywr_model.results['Agr_214IS']['deficit_percent'],
                Pywr_model.results['Agr_220IS']['deficit_percent'],
                Pywr_model.results['Agr_210WB']['deficit_percent'],
                Pywr_model.results['Agr_211WB']['deficit_percent'],
                Pywr_model.results['Agr_212WB']['deficit_percent'],
                Pywr_model.results['Agr_220WB']['deficit_percent'],
                Pywr_model.results['Agr_221WB']['deficit_percent']]) \
                + sum([Pywr_model.results['Mun_210IS']['deficit_percent'],
                Pywr_model.results['Mun_211IS']['deficit_percent'],
                Pywr_model.results['Mun_212IS']['deficit_percent'],
                Pywr_model.results['Mun_214IS']['deficit_percent'],
                Pywr_model.results['Mun_220IS']['deficit_percent'],
                Pywr_model.results['Mun_210WB']['deficit_percent'],
                Pywr_model.results['Mun_211WB']['deficit_percent'],
                Pywr_model.results['Mun_212WB']['deficit_percent'],
                Pywr_model.results['Mun_220WB']['deficit_percent'],
                Pywr_model.results['Mun_221WB']['deficit_percent']])

    # alternative sources have different costs for providing the water'''
    ff_mon = sum([Pywr_model.results['Aschod']['provided_percent'],
            Pywr_model.results['Hadera']['provided_percent'],
                Pywr_model.results['Aschkelon']['provided_percent'],
                Pywr_model.results['Palmachim']['provided_percent'],
                Pywr_model.results['Sorek']['provided_percent']]) * 0.3 \
                + sum([Pywr_model.results['tww_210IS']['provided_percent'],
                Pywr_model.results['tww_211IS']['provided_percent'],
                Pywr_model.results['tww_212IS']['provided_percent'],
                Pywr_model.results['tww_214IS']['provided_percent'],
                Pywr_model.results['tww_220IS']['provided_percent'],
                Pywr_model.results['tww_210WB']['provided_percent'],
                Pywr_model.results['tww_211WB']['provided_percent'],
                Pywr_model.results['tww_212WB']['provided_percent'],
                Pywr_model.results['tww_220WB']['provided_percent'],
                Pywr_model.results['tww_221WB']['provided_percent'] ]) * 0.3 \
                + sum([Pywr_model.results['WFP_external_IS']['provided_percent'],
```

(continues on next page)

(continued from previous page)

```
                Pywr_model.results['WFP_external_WB']['provided_percent']]) * 0.4

    # write the optimization log
    omo.utils.write_logfile(threadnumber,
        [ff_eco+dist, ff_shrt+dist, ff_mon+dist,
        ff_eco, ff_shrt, ff_mon], mf3, parameters)

    print("thread", threadnumber, "dist", dist, "eco", ff_eco, "shrt", ff_shrt, "mon",
↪ ff_mon)

    # dist is the distance to the feasible parameter space. the optimization algorihm␣
↪is allowed
    # to violate this space. parameter values are clipped to space for MODFLOW.
    # the distance acts as a penalty, so the algorithm can learn the feasible␣
↪parameter space.

    return (ff_eco+dist, ff_shrt+dist, ff_mon+dist)
```

The Pywr model in this example operates only on a total balance. Solving the water allocation with discrete time steps is possible. Since the Evaluation of the Pywr results is highly problem specific, therefore you need to write your own code to read and compute the indicators or fitness functions.

2.2.2 Supplementary files

Let's look at the supplementary files.

Well file

Here is a section of the well file.

Index	column	layer	row	p_class	max	min	active_node	well_group
w1	2	245	32	0	-275.406	-2496.742	a220_IS	w220_IS
w2	2	223	36	0	1000	0	a220_IS	w220_IS
w3	2	223	37	0	-107.284	-972.6	a210_IS	w210_IS

The well file specifies several properties of every well with its identifier (*Index*). The location in the MODLOW model is given by *layer*, *row* and *column*. The generic extraction or infiltration pattern *p_class* is defined in the extraction pattern file. As the extraction patterns are scaled between 0.0 and 1.0, the true range for the specific well is recalculated with the *min* and *max* values. Finally, the membership of the well in a *well_goup* enables the sharing of optimization parameters (scaling factors) and *active_node* adds the extractions to an input node in the Pywr model.

The column names are:

1. *Index*... well names
2. *layer*... the layer (z) of the well
3. *row*... the row (y) of the well
4. *column*... the column(x) of the well
5. *p_class*... the extraction pattern, e.g 0 maps to p_class_0 in the extraction pattern file
6. *max*... min and max are used to rescale the extraction pattern to its native range (which is then scaled using the optimization parameter)

7. *min…* see *max*
8. *active_node…* the extractions of this well are added to this Pywr input node
9. *well_group…* the respective well group (share parameters)

Extraction pattern file

Here is a section of the extraction pattern file.

Index	p_class_0	p_class_1
1	1.00E-05	1.01E-05
2	0.082936954	0.048898725
3	0.259970747	0.227219493
4	0.484915378	0.517017079
5	0.733234785	0.80380783
6	0.953547843	0.950135334
7	1	1
8	0.782034395	0.952269837
9	0.465708842	0.83875922
10	0.253697266	0.570840967
11	0.129050065	0.250065387
12	0.036914113	0.056042193

The extraction patterns are given for each season as end of season values. For monthly (30day) and weekly (7day) and daily time steps, the day of year is given at the single well level (see initialization of the Well class). This way, wells can have different season length and the user takes further care of grouping them.

Boundary file

Optimizing the extraction patterns means scaling the complete pattern up or down or use different scaling factors for single seasons. How much a pattern can be scaled up or down for a specific well_goup is defined with the boundary file.

wellgroup	boundary	1	2	3	4	5	6	7	8	9	10	11	12
w210_IS	lower	0.5											
w210_IS	upper	2.0											
w220_IS	lower						0.7	0.8	0.4				
w220_IS	upper						1.3	1.8	1.4				

The extraction pattern for w210_IS can be halved or doubled (from 0.5 to 2.0). For all wells group w220_IS only the seasons 5 to 7 are subject to optimization, all other season are not subject to change.

2.3 Background

Mooflow was developed in the project Medwater. Medwater is part of the funding measure Global Resource Water "GRoW".

Funding number 02WGR1428E B.

2.4 Author

Ruben Müller

Büro für Angewandte Hydrologie, Berlin

software@bah-berlin.de

2.5 License

2.5.1 Software

Mooflow is licensed under the GNU General Public License, Version 3.0 or later.

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 1, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston MA 02110-1301 USA.

2.5.2 Documentation

This documentation is licensed under the GNU Free Documentation License, Version 1.3 or later.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled “GNU Free Documentation License”.

INDEX

Symbols

A

B

C

D

E

F

G

I

L

M

O

P

R

S

T

V

W

Y